A Type-and-Effect System for Object Initialization

FENGYUN LIU, EPFL
ONDŘEJ LHOTÁK, University of Waterloo
AGGELOS BIBOUDIS, EPFL
PAOLO G. GIARRUSSO, Delft University of Technology
MARTIN ODERSKY, EPFL

Every newly created object goes through several initialization states: starting from a state where all fields are uninitialized until all of them are assigned. Any operation on the object during its initialization process, which usually happens in the constructor via *this*, has to observe the initialization states of the object for correctness, i.e. only initialized fields may be used. Checking safe usage of *this* statically, without manual annotation of initialization states in the source code, is a challenge, due to aliasing and virtual method calls on *this*.

Mainstream languages either do not check initialization errors, like Java, C++, Scala, or they defend against them by not supporting useful initialization patterns, such as Swift. In parallel, past research has shown that safe initialization can be achieved for varying degrees of expressiveness but by sacrificing syntactic simplicity.

We approach the problem by upholding *local reasoning* about initialization which avoids whole-program analysis, and we achieve *typestate polymorphism* via subtyping. On this basis, we put forward a novel type-and-effect system that can effectively ensure initialization safety while allowing flexible initialization patterns. We implement an initialization checker in the Scala 3 compiler and evaluate on several real-world projects.

Additional Key Words and Phrases: Object initialization, Type-and-effect system

1 INTRODUCTION

Object-oriented programming is unsafe if objects cannot be initialized safely. The following code shows a simple initialization problem [1]:

```
class Hello {
  val message = "hello, " + name
  val name = "Alice"
}
println(new Hello().message))
```

The code above when run will print “`hello, null`” instead of “`hello, Alice`”, as the field `name` is not initialized, thus holds the value `null`, when it is used in the second line.

The problem of safe initialization comes into existence since the introduction of object-oriented programming, and it is still a headache for programmers and language designers. Joe Duffy, in his popular blog post *on partially constructed objects* [Duffy 2010], wrote:

[1]In the absence of special notes, the code examples are in Scala.

Authors' addresses: Fengyun Liu, EPFL; Ondřej Lhoták, University of Waterloo; Aggelos Biboudis, EPFL; Paolo G. Giarrusso, Delft University of Technology; Martin Odersky, EPFL.

XXXX-XXXX/2020/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Not only are partially-constructed objects a source of consternation for everyday programmers, they are also a challenge for language designers wanting to provide guarantees around invariants, immutability and concurrency-safety, and non-nullability.

1.1 Theoretical Challenges

Checking safe initialization of objects statically is becoming a challenge as the code in constructors is getting more complex. From past research [Fähndrich and Leino 2003; Fähndrich and Xia 2007; Gil and Shragai 2009; Qi and Myers 2009; Servetto et al. 2013; Summers and Müller 2011; Zibin et al. 2012], two initialization requirements are identified and commonly recognized.

Requirement 1: usage of "this" inside the constructor. The usage of already initialized fields in the constructor is safe and supported by almost all industrial languages. Based on an extensive study of over sixty thousand classes, Gil and Shragai [2009] report that over 8% constructors include method calls on `this`. Method calls on `this` can be used to compute initial values for field initialization or serve as a private channel between the superclass and subclass.

Requirement 2: creation of cyclic data structures. Cyclic data structures are common in programming. For example, the following code shows the initialization of two mutually dependent objects:

```
class Parent { val child: Child = new Child(this) }
class Child(parent: Parent)
```

The objective is to allow cyclic data structures while preventing accidental premature usage of aliased objects. Accessing fields or calling methods on those aliased objects under initialization is an orthogonal concern, the importance of which is open to debate.

There are three theoretical challenges to attack the problems above.

Challenge 1: virtual method calls. While direct usage of already initialized fields via `this` is relatively easy to handle, indirect usage via virtual method calls poses a challenge. Such methods could be potentially overridden in a subclass, which makes it difficult to statically check whether it is safe to call such a method. This can be demonstrated by the following example:

```
abstract class AbstractFile {
  def name: String
  val extension: String = name.substring(4)
}
class RemoteFile(url:String) extends AbstractFile {
  val localFile: String = url.hashCode // error
  def name: String = localFile
}
```

According to the semantics of Scala (Java is the same), fields of a superclass are initialized before fields of a subclass, so initialization of the field `extension` proceeds before `localFile`. The field `extension` in the class `AbstractFile` is initialized by calling the abstract method `name`. The latter, implemented in the child class `RemoteFile`, accesses the uninitialized field `localFile`.

Challenge 2: aliasing. It is well-known that aliasing complicates program reasoning and it is challenging to develop practical type systems to support reasoning about aliasing [Clarke et al. 2013; Hogg et al. 1992]. It is also the case for safe initialization: if a field aliases `this`, we may not assume the object pointed to by the field is fully initialized. This can be seen from the following example:

```
class Knot {
  val self = this
  val n: Int = self.n // error
}
```

In the code above, the field `self` is an alias of `this`, thus we may not use it as a fully initialized value. Aliasing may also happen indirectly through method calls, as the following code shows:

```
class Foo {
  def f() = this
  val n: Int = f().n // error
}
```

Challenge 3: typestate polymorphism. Every newly created object goes through several typestates [Strom and Yemini 1986]: starting from a state where all fields are uninitialized until all of them are assigned. If a method does not access any fields on *this*, then it should be able to be called on any typestate of *this*. For example, in the following class `C`, we should be able to call the method `g` regardless of the initialization state of `this`:

```
class C {
  // ...
  def g(): Int = 100
}
```

The challenge is how to support this feature succinctly without syntactic overhead.

1.2 Existing Work

1.2.1 Industrial Languages.

Existing programming languages sit at two extremes. On one extreme, we find languages like Java, C++, Scala, where programmers may use `this` as if it is fully initialized, devoid of any safety guarantee. On the other extreme, we find languages like Swift, which ensures safe initialization, but is overly restrictive. The initialization of cyclic data structures is not supported, calling methods on `this` is forbidden, even the usage of already initialized fields is limited. For example, in the following Swift code, while the usage of `x` to initialize `y` is allowed, the usage of `y` to initialize `f` is illegal, which is a surprise:

```
class Position {
    var x, y: Int
    var f: () -> Int
    init() {
        x = 4
        y = x * x                    // OK
        f = { () -> Int in self.y } // error
    }
}
```

1.2.2 Masked Types.

Masked types [Qi and Myers 2009] is an expressive, flow-sensitive type-and-effect system [Lucassen and Gifford 1988] for safe initialization of objects.

A masked type $T \backslash f$ denotes objects of the type T, where the masked field f cannot be accessed. Each method has an effect signature of the form $\overline{M_1} \rightsquigarrow \overline{M_2}$, which means that the method can only be called if `this` conforms to the masks $\overline{M_1}$, and the resulting masks for `this` after the call is $\overline{M_2}$. However, there are several obstacles to make the system practical.

First, the system incurs cognitive load and syntactic overhead. Many concepts are introduced in the system, such as *subclass masks*, *conditional masks*, *abstract masks*, each with non-trivial syntax. The paper mentions that inference can help to remove the syntactic burden. However, it leaves open the formal development of such an inference system.

Second, the system, while expressive, is insufficient for simple and common use cases due to the missing support for *typestate polymorphism*. This can be seen from the following example, where we want the method g to be called for any initialization state of `this`:

```
class C { def g(): Int = 100 /* effect of g:  ∀M.M ⇝ M  */ }
```

As the method g can be called for `this` with any masks, we would like to give it the (imaginary) polymorphic effect signature $\forall M.M \rightsquigarrow M$, which is not supported. Even if an extension of the system supports polymorphic effect signatures, it will only incur more syntactic overhead.

1.2.3 The Freedom Model. Summers and Müller [2011] propose a light-weight, flow-insensitive type system for safe initialization, which we call *the freedom model*.

The freedom model classifies objects into two groups: *free*, that is under initialization, and *committed*, that is transitively initialized. Field accesses on free objects may get `null`, while committed objects can be used safely. To support typestate polymorphism, it introduces the typestate *unclassified*, which means either *free* or *committed*. With subtyping, typestate polymorphism becomes just *subtyping polymorphism*.

The freedom model supports the creation of cyclic data structures with light-weight syntax. However, the formal system does not address the usage of already initialized fields in the constructor. When an object is free, accessing its field will return a value of the type `unclassified C?`, which means the value could be `null`, free or committed. In the implementation, they introduce *committed-only fields* which can be assumed to be committed with the help of a dataflow analysis. However, the paper leaves open the formal treatment of the dataflow analysis. Our work will address the problem.

Moreover, the abstraction *free* is too coarse for some use cases. This is demonstrated by the following example:

```
1 class Parent {
2   var child = new Child(this)
3   var tag: Int = child.tag     // error in freedom model
4 }
5 class Child(parent: Parent @free) {
6   var tag: Int = 10
7 }
```

According to the freedom model, the expression `child` in line 3 will be typed as *free*, thus the type system cannot tell whether the field `child.tag` is initialized or not. But conceptually we know that all fields of `child` are initialized by the constructor of the class `Child`. In this work we propose a new abstraction to improve expressiveness in such cases.

1.3 Contributions

Our work makes contributions in four areas:

1. Better understanding of local reasoning about initialization. *Local reasoning about initialization* is a key requirement for simple and fast initialization systems. However, while prior work [Summers and Müller 2011] takes advantage of local reasoning about initialization to design simple initialization systems, the concept of local reasoning about initialization is neither mentioned nor defined precisely. Identifying local reasoning about initialization as a concept with a better understanding enables it to be applied in the design of future initialization systems.

2. A more expressive type-based model. We propose a more expressive type-based model for initialization based on the abstractions *cold*, *warm* and *hot*. The introduction of the abstraction *warm* improves the expressiveness of the freedom model [Summers and Müller 2011] which classifies objects as either free (i.e. cold) or committed (i.e. hot).

3. A novel type-and-effect inference system. We propose a type-and-effect inference system for a practical fragment of the type-based model. Existing work usually depends on some unspecified inference or analysis to cut down syntactic overhead [Qi and Myers 2009; Summers and Müller 2011; Zibin et al. 2012]. We are the first to present a formal inference system on the problem of safe initialization. Meanwhile, to our knowledge, we are the first to demonstrate the technique of controlling aliasing in a type-and-effect system.

4. Implementation in Scala 3. We implement an initialization system in the Scala 3 compiler and evaluate it on several real-world projects. The system is capable of handling complex language features, such as inner classes, traits and functions.

2 LOCAL REASONING ABOUT INITIALIZATION

An important insight in the work of Summers and Müller [2011] is that *if a constructor is called with only transitively initialized arguments, the resulting object is transitively initialized.* We give this insight a name, *local reasoning about initialization*; it enables reasoning about initialization without the global analysis of a program, which is the key for simple and fast initialization systems. The insight also holds for method calls: if the receiver and arguments of a method call are transitively initialized, so is the result.

But how can we justify the insight? While a justification can be found in the soundness proof of the freedom model, it is obscured in a monolithic proof structure (see Lemma 1 of Summers and Müller [2011]). We provide a modular understanding of local reasoning about initialization by identifying three semantic properties, which we call *weak monotonicity*, *stackability* and *scopability*. Identifying local reasoning about initialization as a concept with a better understanding enables it to be applied in the design of future initialization systems. The properties can be explained roughly as follows:

- weak monotonicity: initialized fields continue to be initialized.
- stackability: all fields of a class should be initialized at the end of the class constructor.
- scopability: objects under initialization can only be accessed via static scoping.

To study the properties more formally, we first introduce a small language.

2.1 A Small Language

Our language resembles a subset of Scala having only top-level classes, mutable fields and methods.

$$\begin{array}{rcl}
\mathcal{P} \in \text{Program} & ::= & (\overline{C}, D) \\
C \in \text{Class} & ::= & \mathit{class}\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
\mathcal{F} \in \text{Field} & ::= & \mathit{var}\ f{:}T = e \\
e \in \text{Exp} & ::= & x \mid \mathit{this} \mid e.f \mid e.m(\overline{e}) \mid \mathit{new}\ C(\overline{e}) \mid e.f = e; e \\
\mathcal{M} \in \text{Method} & ::= & \mathit{def}\ m(\overline{x{:}T}) : T = e \\
S, T, U \in \text{Type} & ::= & C
\end{array}$$

A program $\mathcal{P}$ is composed of a list of class definitions and an entry class. The entry class must have the form $\mathit{class}\ D\ \{\ \mathit{def}\ \mathit{main} : T = e\ \}$. The program runs by executing e.

A class definition contains class parameters ($\hat{f}{:}T$), field definitions ($\mathit{var}\ f{:}T = e$) and method definitions. Class parameters are also fields of the class. All class fields are mutable. As a convention, we use f to range over all fields, and $\hat{f}$ to only range over class parameters.

An expression (e) can be a variable (x), self reference (this), field access ($e.f$), method call ($e.m(\overline{e})$), class instantiation ($\mathit{new}\ D(\overline{e})$), block expression ($e.f = e; e$). The block expression is used to avoid introducing the syntactic category of statements in the presence of assignments, which simplifies the presentation and meta-theory.

A method definition is standard. We restrict method body to just expressions. This choice simplifies the meta-theory without loss of expressiveness thanks to block expressions.

The following constructs are used in defining the semantics:

$$\begin{aligned}
\Xi \in \text{ClassTable} &= \text{ClassName} \rightharpoonup \text{Class}\\
\sigma \in \text{Store} &= \text{Loc} \rightharpoonup \text{Obj}\\
\rho \in \text{Env} &= \text{Variable} \rightharpoonup \text{Value}\\
o \in \text{Obj} &= \text{ClassName} \times (\text{FieldName} \rightharpoonup \text{Value})\\
l, \psi \in \text{Value} &= \text{Loc}
\end{aligned}$$

We use ψ to denote the value of *this*, σ corresponds to the heap, ρ corresponds to the local variable environment of the current stack frame.

The big-step semantics, presented in Appendix C is standard, thus we omit detailed explanation. The only note is that non-initialized fields are represented by missing keys in the object, instead of a *null* value. Newly initialized objects have no fields, and new fields are gradually inserted during initialization until all fields defined by the class have been assigned.

Note that this language does not enjoy initialization safety, and it is the task of later sections to make it safe. However, the language enjoys local reasoning about initialization.

2.2 Definitions

Definition 2.1 (reachability). An object l' is reachable from l in the heap σ, written $\sigma \vDash l \rightsquigarrow l'$, is defined below:

$$\frac{l \in dom(\sigma)}{\sigma \vDash l \rightsquigarrow l} \qquad \frac{\sigma \vDash l_0 \rightsquigarrow l_1 \qquad (\_, \omega) = \sigma(l_1) \qquad \exists\, f.\ \omega(f) = l_2 \qquad l_2 \in dom(\sigma)}{\sigma \vDash l_0 \rightsquigarrow l_2}$$

Definition 2.2 (reachability for set of locations).

$$\begin{aligned}
\sigma \vDash L \rightsquigarrow l &\triangleq \exists l' \in L. \sigma \vDash l' \rightsquigarrow l\\
\sigma \vDash l \rightsquigarrow L &\triangleq \exists l' \in L. \sigma \vDash l \rightsquigarrow l'
\end{aligned}$$

Definition 2.3 (cold). An object is cold if it exists in the heap, formally

$$\sigma \vDash l : cold \quad \triangleq \quad l \in dom(\sigma)$$

Definition 2.4 (warm). An object is warm if all its fields are assigned, formally

$$\sigma \vDash l : warm \quad \triangleq \quad \exists (C, \omega) = \sigma(l) \bigwedge fields(C) \subseteq dom(\omega)$$

Definition 2.5 (hot). An object is hot if all reachable objects are warm, formally

$$\sigma \vDash l : hot \quad \triangleq \quad l \in dom(\sigma) \bigwedge \forall l'. \sigma \vDash l \rightsquigarrow l' \Longrightarrow \sigma \vDash l' : warm$$

From the definition, it is easy to see that *hot* implies *warm* and *warm* implies *cold*.

2.3 Weak Monotonicity

The idea of *monotonicity* dates back to *heap monotonic typestates* by Fähndrich and Leino [2003]. There are, however, at least three different concepts of monotonicity.

Weak monotonicity means that initialized fields continue to be initialized. More formally, we may prove the following theorem:

Theorem 2.6 (Weak Monotonicity).

$$[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \Longrightarrow \sigma \preceq \sigma'$$

In the above, the predicate *weak monotonicity* ($\sigma \preceq \sigma'$) is defined below:

Definition 2.7 (Weak Monotonicity).

$$\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).\ (C, \omega) = \sigma(l) \quad \Longrightarrow \quad (C, \omega') = \sigma'(l)$$

While weak monotonicity is sufficient to justify local reasoning about initialization, stronger monotonicity is required for initialization safety. For example, the freedom model [Summers and Müller 2011] enforces *strong monotonicity*:

$$\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).\ \sigma \vDash l : \mu \Longrightarrow \sigma' \vDash l : \mu$$

In the above, we abuse the notation by using μ to denote either *cold*, *warm* or *hot*. Strong monotonicity additionally ensures that hot objects continue to be hot. Therefore, it is always safe to use hot objects freely. However, to enforce safer usage of already initialized fields of non-hot objects, we need an even stronger concept, *perfect monotonicity*:

$$\begin{aligned}\sigma \preceq \sigma' \quad \triangleq \quad \forall l \in dom(\sigma).\quad & (C, \omega) = \sigma(l) \Longrightarrow \\ & (C, \omega') = \sigma'(l) \quad \bigwedge \quad \forall f \in dom(\omega). \sigma \vDash \omega(f) : \mu \Longrightarrow \sigma' \vDash \omega'(f) : \mu\end{aligned}$$

In the above, we abuse the notation by writing directly $\omega'(f)$ to require that $dom(\omega) \subseteq dom(\omega')$. Perfect monotonicity in addition ensures that initialization states of object fields are monotone. It will be problematic if a field is initially assigned a hot value and later reassigned to a non-hot value.

2.4 Stackability

Conceptually, stackability ensures that all newly created objects during the evaluation of an expression e are *warm*, i.e. all fields of the objects are assigned. Formally, the insight can be proved as a theorem:

Theorem 2.8 (Stackability).

$$[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \Longrightarrow \sigma \ll \sigma'$$

The predicate $\sigma \ll \sigma'$ is defined below, which says that for any object in the heap σ', either the object is *warm*, or the object pre-exists in the heap σ.

Definition 2.9 (Stacking).

$$\sigma \ll \sigma' \quad \triangleq \quad \forall l \in dom(\sigma'). \sigma' \vDash l : warm \bigvee l \in dom(\sigma)$$

Definite assignment [Gosling et al. 2015] can be used to enforce stackability in programming languages. In Java, however, it only requires that final fields are initialized.

If we push an object in a stack when it comes into existence, and remove it from the stack when all its fields are assigned, we will find that the object to be removed is always at the top of the stack. This is illustrated in Figure 1.

2.5 Scopability

Scopability says that the access to uninitialized objects should be controlled by static scoping. Intuitively, it means that a method may only access pre-existing uninitialized objects through its environment, i.e. method parameters and `this`.

Objects under initialization are dangerous when used without care, therefore the access to them should be controlled. Scopability imposes discipline on accessing uninitialized objects. If we regard

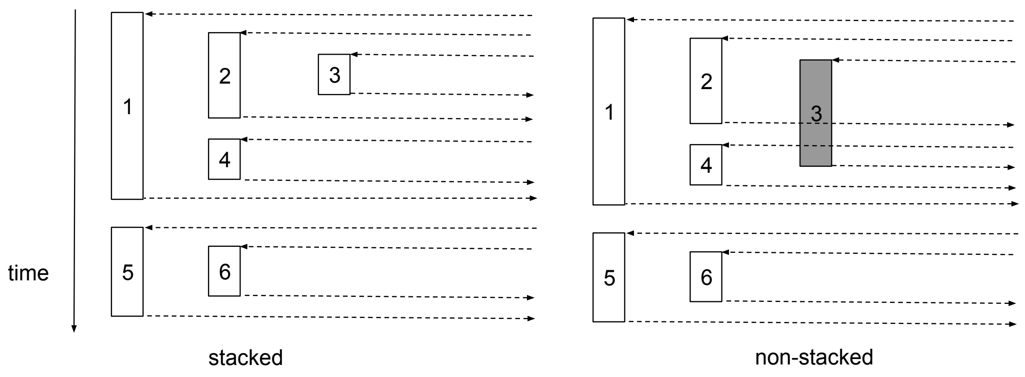

Fig. 1. Each block represents the initialization duration of an object, i.e., from the creation of the object to the point where all fields are assigned.

uninitialized objects as capabilities, then scopability restricts that there should be no side channels for accessing those capabilities. All accesses have to go through the explicit channel, i.e. method parameters and `this`. In contrast, global variables or control-flow effects such as algebraic effects may serve as side channels for teleporting values under initialization. To maintain local reasoning about initialization, an initialization system needs to make sure that only initialized values may travel by side channels.

More formally, we can prove the following theorem:

Theorem 2.10 (Scopability).

$$[\![e]\!]\,(\sigma, \rho, \psi) = (l, \sigma') \implies (\sigma, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma', \{\ l\ \}) \tag{1}$$

In the above, the predicate $(\sigma, L) \lessdot (\sigma', L')$ is defined below:

Definition 2.11 (Scoping). A set of addresses $L' \subseteq dom(\sigma')$ is *scoped* by a set of addresses $L \subseteq dom(\sigma)$, written $(\sigma, L) \lessdot (\sigma', L')$, is defined as follows

$$(\sigma, L) \lessdot (\sigma', L') \quad \triangleq \quad \forall l \in dom(\sigma). \quad \sigma' \vDash L' \rightsquigarrow l \implies \sigma \vDash L \rightsquigarrow l$$

The theorem means that if e evaluates to l, then l and every location l' reachable from l in the new heap is either fresh, in that it did not exist in the old heap, or it *was* reachable from $codom(\rho) \cup \psi$ in the *old heap*.

Note that in the definition of *scoping*, we use $\sigma \vDash L \rightsquigarrow l$ instead of $\sigma' \vDash L \rightsquigarrow l$. This is because in a language with mutation, l may no longer be reachable from L in σ' due to reassignment. This can be seen in Figure 2.

The property of scopability holds intuitively, but its proof is not obvious at all. The subtlety is in proving the case $e_1.m(e_2)$. Suppose we have $[\![e_1]\!]\,(\sigma_1, \rho, \psi) = (l_1, \sigma_2)$ and $[\![e_2]\!]\,(\sigma_2, \rho, \psi) = (l_2, \sigma_3)$. By induction hypothesis, we have $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_2, l_1)$ and $(\sigma_2, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_3, l_2)$. However, we do not know that $(\sigma_1, codom(\rho) \cup \{\ \psi\ \}) \lessdot (\sigma_3, l_1)$. We need some invariant saying that scoping relations are preserved. That invariant has to be carefully defined, as not all scoping relations are preserved due to reassignment. We refer the reader to the technical report for more detailed discussions [Liu et al. 2020].

2.6 Local Reasoning about Initialization

With weak monotonicity, stackability and scopability, we may prove the theorem of local reasoning about initialization.

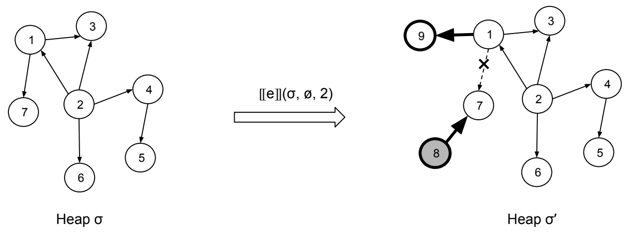

Fig. 2. Each circle represents an object and numbers are locations. An arrow means that an object holds a reference to another object. The thick circles and links on the right heap are new objects and links created during the execution. Due to scopability, we have $(\sigma, \{2\}) \lessdot (\sigma', \{8\})$. It means if the result object 8 reaches any object which pre-exists in the heap σ, then the object must be reachable from object 2 in the heap σ. The object 7 which is reachable from the object 2 in the heap σ, is no longer reachable from object 2 in the heap σ' due to the removal of the link from object 1 to object 7.

Lemma 2.12 (Local Reasoning). *The following proposition holds*

$$\frac{(\sigma, L) \lessdot (\sigma', L') \qquad \sigma \ll \sigma' \qquad \sigma \preceq \sigma' \qquad \sigma \vDash L : hot}{\sigma' \vDash L' : hot}$$

Proof. Let's consider a reachable object l from L', i.e. $\sigma' \vDash L' \rightsquigarrow l$. Depending on whether $l \in dom(\sigma)$, there are two cases.

- **Case** $l \notin dom(\sigma)$.
  Use the fact that $\sigma \ll \sigma'$, we know $\sigma' \vDash l : warm$.

- **Case** $l \in dom(\sigma)$.
  Use the fact that $(\sigma, L) \lessdot (\sigma', L')$, we have $\sigma \vDash L \rightsquigarrow l$. From the premise $\sigma \vDash L : hot$, we have $\sigma \vDash l : warm$. From $\sigma \preceq \sigma'$, we have $\sigma' \vDash l : warm$.

In both cases, we have $\sigma' \vDash l : warm$, by definition we have $\sigma' \vDash L' : hot$. □

Theorem 2.13 (Local Reasoning). *The following proposition holds:*

$$\frac{[\![e]\!](\sigma, \rho, \psi) = (l, \sigma') \qquad \sigma \vDash \{\ \psi\ \} \cup codom(\rho) : hot}{\sigma' \vDash l : hot}$$

Proof. Immediate from Lemma 2.12, the preconditions are satisfied by Theorem 2.10, Theorem 2.6 and Theorem 2.8. □

In particular, if e is a method body, we can conclude that if the receiver and all the method parameters are hot, then the return value is also hot.

This theorem echoes the insight in the freedom model [Summers and Müller 2011]: if a constructor is called with all arguments committed, then the constructed object is also committed.

3 THE BASIC MODEL

In this section, we take advantage of local reasoning about initialization to develop a type system that ensures initialization safety of objects.

3.1 Types

From the last section, we see that there are three natural abstractions of initialization states:

cold	A cold object *may* have uninitialized fields.
warm	A warm object has all its fields initialized.
hot	A hot object has all its fields initialized and only reaches hot objects.

If we posit the abstractions *cold*, *warm* and *hot* as types, we arrive at a type system for safe initialization of objects, which we call *the basic model*. Types in the language have the form C^{μ}:

$$\begin{array}{rcl} \Omega & ::= & \{ f_1, f_2, \ldots \} \\ \mu & ::= & cold \mid warm \mid hot \mid \Omega \\ T & ::= & C^{\mu} \end{array}$$

The type C^{Ω} is introduced to support the usage of already initialized fields — Ω denotes the set of initialized fields. The type is well-formed if Ω contains only fields of the class C. In languages that are equipped with an annotation system, such as Java, the type C^{μ} can be written using annotations (e.g. C *@warm* and C *@cold*), while a type without annotation can be assumed to be *hot*. Types like C^{Ω} are mainly used internally in the type system, thus there is no need to write them explicitly.

A type C^{μ_1} is a subtype of another type C^{μ_2}, written $C^{\mu_1} <: C^{\mu_2}$, if $\mu_1 \sqsubseteq \mu_2$. The lattice for modes μ is defined below:

$$hot \sqsubseteq \mu \qquad warm \sqsubseteq \Omega \qquad \Omega_1 \cup \Omega_2 \sqsubseteq \Omega_1 \qquad \mu \sqsubseteq cold$$

The modes *hot* and *cold* are respectively bottom and top of the lattice, and Ω is in the middle.

Methods are now annotated with modes, i.e., in $@\mu\ def\ m(\overline{x{:}T}) : T = e$, the mode μ means *this* has the type C^{μ} inside the method m of the class C. We will propose an inference system to avoid the annotations (Section 4 and 5). The semantics of the language remain the same as the language introduced in section 2.

3.2 Type System

We present definition typing and expression typing in Figure 3 and Figure 4. In an expression typing judgment $\Gamma; T \vdash e : U$, the type T is the type for *this*. Note that for simplicity of presentation, the class table Ξ is omitted in expression typing judgments.

- The rule T-Block demands that we only reassign *hot* values to fields; that is how we enforce *perfect monotonicity* in the system.
- Both the rules T-New and T-Invoke take advantage of *local reasoning about initialization* at the type level.
- The rule T-SelHot capitalizes on the fact that a hot object may only reach hot objects.

The soundness theorem says that a well-typed program does not get stuck at runtime.

Theorem 3.1 (Soundness). *If* $\vdash \mathcal{P}$, *then* $\forall k.\ [\![\mathcal{P}]\!](k) \neq Error$

The meta-theory takes the approach of step-indexed definitional interpreters [Amin and Rompf 2017]. For a step-indexed interpreter, there are three possible outcomes: (1) time out; (2) error; (3) a resulting value and an updated heap. Initialization safety is implied by soundness, as initialization errors will cause the program to fail at runtime. We refer the reader to the technical report for more details about the meta-theory [Liu et al. 2020].

Program Typing $\boxed{\vdash \mathcal{P}}$

$$\frac{\Xi = \overline{C \to C} \qquad \Xi(D) = class\ D\ \{\ def\ main : T = e\ \} \qquad \emptyset; D^{hot} \vdash e : T \qquad \overline{\Xi \vdash C}}{\vdash (\overline{C}, D)}\ \ (\text{T-Prog})$$

Class Typing $\boxed{\Xi \vdash C}$

$$\frac{\Omega_0 = \overline{\hat{f}} \qquad \overline{\Xi; C^{\Omega_i} \vdash \mathcal{F}_i \quad \Omega_{i+1} = \Omega_i \cup \{\ f_i\ \}} \qquad \overline{\Xi; C \vdash \mathcal{M}}}{\Xi \vdash class\ C(\overline{\hat{f}:T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}}\ \ (\text{T-Class})$$

Field Typing $\boxed{\Xi; C^{\Omega} \vdash \mathcal{F}}$

$$\frac{\emptyset; C^{\Omega} \vdash e : T}{\Xi; C^{\Omega} \vdash var\ f : T = e}\ \ (\text{T-Field})$$

Method Typing $\boxed{\Xi; C \vdash \mathcal{M}}$

$$\frac{\overline{x:T}; C^{\mu} \vdash e : S}{\Xi; C \vdash @\mu\ def\ m(\overline{x:T}) : S = e}\ \ (\text{T-Method})$$

Fig. 3. Definition typing of the basic model

3.3 Typestate Polymorphism and Authority

A key design decision of the type system is to embrace *flow-insensitivity*. It follows an insight from Summers and Müller [2011] that we may achieve *typestate polymorphism* via subtyping in a flow-insensitive system.

Otherwise, if the system were flow-sensitive, we would have to track the change of typestates of `this` inside a method. Suppose we track the changes of a method `m` with $C^{\mu_1} \to C^{\mu_2}$, which means that the method `m` requires `this` to conform to C^{μ_1} before the call, and `this` takes the typestate C^{μ_2} after the call, similar to what is done by Qi and Myers [2009]. This creates a difficulty for methods that can be called for any typestates of `this`, as the following example shows:

```
class C {
  // ...
  def g(): Int = 100 //  ∀μ.C^μ → C^μ
}
```

In the code above, the method `g` can be called for any typestate of `this`. Representing the fact in the system would require *parametric polymorphism*, which complicates the solution. In fact, the system proposed by Qi and Myers [2009] does not support typestate polymorphism and thus invalidates such simple use cases.

The combination of flow-insensitivity and strong/perfect monotonicity imposes a rule in the design of initialization systems, which we call *authority*. Roughly, it means that we may only advance initialization states of an object at specific locations in its class constructor.

Expression Typing $\boxed{\Gamma; T \vdash e : T}$

$$\frac{\Gamma; T \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma; T \vdash e : T_2} \quad \text{(T-SUB)}$$

$$\frac{x : U \in \Gamma}{\Gamma; T \vdash x : U} \quad \text{(T-VAR)}$$

$$\Gamma; T \vdash \mathit{this} : T \quad \text{(T-THIS)}$$

$$\frac{\Gamma; T \vdash e : D^{hot} \qquad C^{\mu} = \mathit{fieldType}(D, f)}{\Gamma; T \vdash e.f : C^{hot}} \quad \text{(T-SELHOT)}$$

$$\frac{\Gamma; T \vdash e : D^{warm} \qquad U = \mathit{fieldType}(D, f)}{\Gamma; T \vdash e.f : U} \quad \text{(T-SELWARM)}$$

$$\frac{\Gamma; T \vdash e : D^{\Omega} \qquad f \in \Omega \qquad U = \mathit{fieldType}(D, f)}{\Gamma; T \vdash e.f : U} \quad \text{(T-SELOBJ)}$$

$$\frac{\overline{T_i} = \mathit{constrType}(C) \qquad \Gamma; T \vdash e_i : C_i^{\mu_i} \qquad C_i^{\mu_i} <: T_i \qquad \mu = (\sqcup\, \mu_i) \sqcap \mathit{warm}}{\Gamma; T \vdash \mathit{new}\ C(\overline{e}) : C^{\mu}} \quad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma; T \vdash e : C^{\mu_0} \qquad (\mu_m, \overline{T_i}, D^{\mu_r}) = \mathit{methodType}(C, m) \\ \mu_0 \sqsubseteq \mu_m \qquad \Gamma; T \vdash e_i : D_i^{\mu_i} \qquad D_i^{\mu_i} <: T_i \qquad \mu = (\sqcup\, \mu_i = \mathit{hot})?\mathit{hot} : \mu_r\end{array}}{\Gamma; T \vdash e.m(\overline{e}) : D^{\mu}} \quad \text{(T-INVOKE)}$$

$$\frac{\Gamma; T \vdash e_1.f : C^{\mu} \qquad \Gamma; T \vdash e_2 : C^{hot} \qquad \Gamma; T \vdash e : T_1}{\Gamma; T \vdash e_1.f = e_2; e : T_1} \quad \text{(T-BLOCK)}$$

Fig. 4. Expression typing of the basic model

In a flow-insensitive system, how can we safely advance typestates of objects? It is unsafe to do so at arbitrary locations in the program, as the update may break monotonicity if the typestate of the object can be advanced differently via aliases elsewhere. More theoretically, a flow-insensitive system cannot establish the order of updates at different locations (and possibly via aliases) to guarantee monotonicity.

The rule of authority suggests that it is safe to perform typestate updates only via an outstanding alias and only at definite locations in a local context. The definite locations form a *local flow* in a flow-insensitive system. In the experimental language, the outstanding alias is `this`, and the locations are the points of field initializations inside the class constructor.

The design rule of authority comes from the meta-theory. The meta-theory is based on *store typing* [Pierce 2002, Chapter 13]. We use Σ to range over store typings, which are maps from locations to types, i.e. $Loc \rightharpoonup Type$. An important semantic property used in the proof is the following:

$$\frac{\forall l \in dom(\Sigma).\Sigma(l) = C^{\Omega} \implies \Sigma'(l) = C^{\Omega}}{\Sigma \rhd \Sigma'}$$

In the above, Σ and Σ' are the store typings before and after evaluating an expression. Literally, the property says that if the object at location l is considered to have initialization state Ω before the evaluation of an expression, it must be considered to still have the initialization state Ω after the evaluation of the expression.

Intuitively, the property implies that we may not advance the initialization state of existing objects during evaluation of an expression. It leaves only the possibility to advance object state at special locations in the constructor. At the end of the class body when all fields are initialized, we promote the type of the fresh object to be `warm`. Its promotion to `hot` may be delayed until a group of cyclic objects becomes hot together, which is called a *commitment point* by Summers and Müller [2011].

We call the semantic property above the *property of authority*. The property is necessary to prove *perfect monotonicity*, which is an important invariant in the soundness proof. The reason can be demonstrated by the following program:

```
class C(a: A @cold) { var x: D @cold = e; var y: Int = 10 }
```

In the code above, suppose the type of ψ (the value for `this`) starts as $C^{\{ a \}}$, and a side-effect of evaluating e updates the type of ψ to C^{μ}. After assigning the value of e, denoted as l_e, to the field x, we update the type of `this` to $C^{\mu'}$. We would like $\mu' \sqsubseteq \{ a, x \}$ to record that the fields a and x is initialized, and monotonicity requires that $\mu' \sqsubseteq \mu$. The property of authority ensures that $\mu = \{ a \}$, which enables one simple solution to these constraints, namely $\mu' = \{ a, x \}$. This is a sound choice because we do know that the field x has been assigned the value l_e, which is of the type D^{cold} (known by induction hypothesis) as required by the semantic typing of the object ψ as $C^{\{ a,x \}}$.

In the absence of authority, it would be allowed to update the type of ψ as a side-effect of evaluating e, for example to C^{hot}. Then the constraint $\mu' \sqsubseteq \mu$ would force μ to be *hot*. However, there is no guarantee that l_e is transitively initialized. From the induction hypothesis, we only know that it has the type D^{cold}. So setting μ' to *hot* would be unsound, since `this` might no longer be transitively initialized after the field x is assigned the value l_e.

Note that the property of authority only talks about types of the form C^{Ω}. The store typing never contains types like C^{cold}, a value takes such a type by subtyping. Authority for values of the type C^{warm} is not necessary for soundness. The reason is that the only operation in the store typing for a warm object is to promote its type from C^{warm} to C^{hot}, the only possible next monotone state, it is thus impossible for monotonicity to fail. For the type C^{hot}, monotonicity guarantees that the type keeps the same.

We believe the property of authority is already necessary for a flow-insensitive system that enforces strong monotonicity, such as the freedom model [Summers and Müller 2011], but it has not been made explicit in previous work.

4 TYPE-AND-EFFECT INFERENCE, INFORMALLY

The type system proposed in the last section depends on verbose annotations, which forms an obstacle for its adoption in practice. In this section, we propose a type-and-effect inference system [Lucassen and Gifford 1988; Nielson et al. 1999] to significantly cut down the syntactic overhead. We first discuss the design of the type-and-effect inference system informally by examples.

4.1 Potentials and Effects

Consider the following erroneous program, which accesses the field y before it is initialized:

```
class Point {
    var x: Int = this.y      // Point.this.y!
    var y: Int = 10
}
```

A natural idea to ensure safe initialization is to analyze the fields that are accessed at each step of initialization, and check that only initialized fields are accessed. This leads to the fundamental effect in initialization: **field access effect**, e.g. $C.this.f!$.

Fields may also be accessed *indirectly* through method calls, as the following code shows:

```
class Point {
    var x: Int = this.m()  // Point.this.m<>
    var y: Int = 10
    def m(): Int = this.y  // Point.this.y!
}
```

For this case, we may introduce method calls as effects, which act as placeholders for the actual effects that happen in the method: **method call effects**, e.g. $C.this.m\Diamond$.

If we first analyze effects of the method m and map the effect $Point.this.m\Diamond$ to the set of effects $\{Point.this.y!\}$, then we may effectively check the initialization error in the code above.

One subtlety in the design is how to handle aliasing. We illustrate with the following example:

```
class Knot {
    var self  = this      // potentials of "self": { Knot.this }
    var x: Int = self.x   // effects of "self.x": { Knot.this.self.x! }
}
```

In the code above, the field `x` is used via the alias `self` before it is initialized. To check such errors, we need a way to represent the aliasing information in the system. That leads us to the concept of **potentials**. Potential over-approximate expressions that could potentially alias a potentially uninitialized object. If an expression could refer to an uninitialized object, it must be abstracted by a potential. If an expression has an empty set of potentials, it means at runtime the value of the expression is always hot.

A potential encodes aliasing information in the form of paths, such as `C.this`, `C.this.f` or `C.this.m`. In the code example above, the field `self` takes the potentials of its initializer, i.e. the set `{ Knot.this }`. Now an initialization checker may take advantage of the aliasing information and report an error for the code `self.x`.

To enforce that all arguments to method calls are hot, we introduce **promotion effects** that promote potentials to be hot, e.g. $C.this\uparrow$. The checking system will check that only hot objects are promoted. The following example illustrates the usage of the effect:

```
class Point {
    var x: Int = this.m()       //  Point.this.m<>
    def m(): Int = call(this)   //  Point.this↑
}
```

In the code above, the method call effect $Point.this.m\Diamond$ incurs the promotion effect $Point.this\uparrow$. The system finds that at the point of the call `this.m()`, the value of `this` is not hot, such promotion is thus illegal.

Semantically, potentials keep track of objects possibly under initialization in order to maintain a *directed segregation* of initialized objects and objects under initialization: objects under initialization may point to initialized objects, but not vice versa. A promotion effect means that the object pointed to by the potential ascends to the initialized world, and the system gives up on tracking it. The system will have to ensure that by the time this happens, the object is hot.

Note that field access $C.this.a!$ and field promotion $C.this.a\uparrow$ are different effects, because field access does not necessarily need to promote the field, as demonstrated by the following example:

```
class Knot {
  var a = this
  var b = this.a // Knot.this.a! , but no promotion
}
```

Aliasing and promotion may also happen through methods, as the following example shows:

```
1 class Fact {
2     var value = escape(this.m()) //  Fact.this.m↑
3     def m() = this               //  potentials of m: { Fact.this }
4 }
```

The type-and-effect system knows that the return value of the method `m` aliases `this`, thus the promotion of `this.m()` at line 2 indirectly promotes `this`.

A similar distinction is drawn on methods: (1) the method invocation effect $C.this.m\Diamond$ means that the method `m` is called with the receiver `this`; (2) the method promotion effect $C.this.m\uparrow$ means that the return value of the call `this.m` is promoted.

4.2 Two-Phase Checking

A common issue in program analysis is how to deal with recursive methods. We tackle the problem with *two phase checking*. In the first phase, the system computes effect summaries for methods and fields. In the second phase, the system checks that no fields are used before they are initialized. During the checking, it uses the effect summaries from the first phase. For example, assume the following program:

```
class Foo {
  var a: Int = h()
  def h(): Int = g()
  def g(): Int = h()
}
```

In the first phase, the computed summary for the methods `h` and `g` is as follows:

method	effects	potentials
h	$\{ Foo.this.g\Diamond \}$	$\{ Foo.this.g \}$
g	$\{ Foo.this.h\Diamond \}$	$\{ Foo.this.h \}$

In the second phase, while checking the method call `h()`, the analysis propagates the effects associated with the method `h` until it reaches the fixed point $\{ Foo.this.g\Diamond, Foo.this.h\Diamond \}$. As the set does not contain accesses to any uninitialized fields of `this` nor invalid promotion, the program passes the check. Note that the domain of effects has to be finite for the existence of the fixed point.

4.3 Full-Construction Analysis

Another common issue in analysis is how to handle virtual method calls. The approach we take is *full-construction analysis*: we treat the constructors of concrete classes as entry points, and check all super constructors as if they were inlined. The analysis spans the full duration of object construction. This way, all virtual method calls on `this` can be resolved statically. From our experience, full-construction analysis greatly improves user experience, as no annotations are required for the interaction between subclasses and superclasses.

The following problem also motivates us to check the full construction duration of an object, which is also known as the *fragile base class problem*:

```
class Base { def g(): String = "hello" }
class Foo extends Base { val a = this.g() }
class Bar extends Base {
  val b: String = "b"
  override def g(): String = this.b
}
```

This program is correct. However, if we follow a type-based approach like the freedom model [Summers and Müller 2011], in order to call `g()` in the class `Foo`, the method `Base.g` has to be annotated `@free`, so that it may not access any fields on `this`. For soundness, the overriding method `Bar.g` has to be annotated `@free` too: but now it may not access the field `this.b` in the body of the method `Bar.g`. This unnecessarily restricts expressiveness of the system.

Moreover, we believe it is the only practical way to handle complex language features such as properties and traits. In languages such as Scala and Kotlin, fields are actually properties, accesses of public field are dynamic method calls, as the following code shows:

```
class A { val a = "Bonjour"; val b: Int = a.size }
class B extends A { override val a = "Hi" }
new B
```

In the code above, when the constructor of class `B` calls the constructor of class `A`, the expression `a.size` will dynamically dispatch to read the field `a` declared in class `B`, not the field `a` declared in class `A`. This results in a null-pointer exception at runtime because at the time the field `a` in class `B` is not yet initialized. Without full-construction analysis, it is difficult to make the analysis sound for the code above.

Closed World Assumption. Full-construction analysis does not assume a *closed world* in the sense that it does not depend on the program entry as the analysis entry point. In contrast, it takes constructors of concrete classes as analysis entry points. In the analysis, it requires the code of constructors of superclasses to be available.

Modularity. While full-construction analysis is capable of handling language features like traits and properties, it pays the price of modularity in the sense that if a superclass is changed, the subclasses have to be recompiled. We believe this is a worthy price to pay. First, the coupling between super class and subclass is well-known in object-oriented programming. For example, if a superclass adds a new method, then all its subclasses have to be recompiled to check proper overriding. Second, the ideal granularity for modular checking is not classes, but projects. From our experience with real-world projects, most subtle initializations happen within the same project. Third, the type system presented in Section 3 can serve as coarse-grained type specification at project boundaries.

4.4 Cyclic Data Structures

Cyclic data structures are supported with an annotation *@cold* on class parameters, as the following example demonstrates:

```
class Parent { val child: Child = new Child(this) }
class Child(parent: Parent @cold) {
  val friend: Friend = new Friend(this.parent)
}
class Friend(parent: Parent @cold) { val tag = 10 }
```

The annotation `@cold` indicates that the actual argument to `parent` during object construction might not be initialized. The type-and-effect system will ensure that the field `parent` is not used directly or indirectly when instantiating `Child`. However, aliasing the field to another cold class parameter is fine, thus the code `new Friend(this.parent)` at line 3 is accepted by the system. This allows programmers to create complex aliasing structures during initialization.

Our system tracks the return value of `new Child(this)` as the set of potentials $\{ \mathit{warm}[\mathit{Child}] \}$. All fields of a warm value are assigned, but they may hold values that are not fully initialized. The inference system also takes advantage of local reasoning about initialization (Section 2): the whole cyclic data structure becomes hot at the same time when the first object in the group, i.e. the instance of `Parent`, becomes warm. This is called *commitment point* in the work of Summers and Müller [2011].

4.5 Relationship with the Type System

The type-and-effect system is intended to serve as an inference system for the type system in Section 3. Although simpler, the type system there requires annotations and thus forms an obstacle for adoption in practice. Meanwhile, the type-and-effect system scales better to complex language features like properties, inner classes and functions, and integrates better with compilers as no changes to the type system of the compiler are needed.

That said, the type-and-effect system is based on the type system in Section 3, and can be regarded as an inference system for a fragment of the type system there. For example, given the following code:

```
class C {
  val d: D = new D(this)
  def foo = this.n
  foo
  val n = 10
}
class D(c: C @cold) {
  val tag = 10
}
```

The field `d` is associated with the potentials $\{ \mathit{warm}[D] \}$, it may thus take the type D^{warm}. The method `foo` is associated with the effects $\{ \mathit{this}.n! \}$, which suggests that `this` should conform to the type $C^{\{ n \}}$ when the method `foo` is called.

In practice, the type-and-effect system does not bother to compute the exact type annotations nor elaborate the program with such type annotations, because the type elaboration is not useful in later compiler phases. Instead, it only checks that all the effects are safe in the constructor.

The fragment of the type system that we identify demands that (1) *method arguments must be hot*, and (2) *non-hot class parameters must be annotated*. The fragment supports calling methods on *this* in the constructor, as well as creation of cyclic data structures. There are several considerations for the restrictions.

First, from practical experience, there is little need to use non-hot values as method arguments. Meanwhile, virtual method calls on `this` are allowed, which covers most use cases in practice [Gil and Shragai 2009].

Second, it agrees with good programming practices that values under initialization should not escape [Bloch 2008]. Therefore, when there is the need to pass non-hot arguments to a constructor, it is a good practice to mark them explicitly.

Third, demanding method arguments to be hot saves us from changing the core type system of a language to check safe overriding of virtual methods.

5 FORMALIZING TYPE-AND-EFFECT INFERENCE

In this section, we formalize the type-and-effect system presented informally in the last section. Due to space limit, the soundness proof of the system is presented in the technical report included as supplemental material.

5.1 Syntax and Semantics

Our language is almost the same as the language introduced in section 2, except for the definition of class parameters. In a class definition like *class* $C(\overline{\hat{f}{:}T})$ $\{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}$, we introduce *cold class parameters*, which has the syntax $\tilde{f}$. Cold class parameters may take a value that is not transitively initialized. A class parameter $\hat{f}$ is also a field of its defining class. By default, we use f to range over all fields, $\hat{f}$ over class parameters, and $\tilde{f}$ over cold class parameters.

The tilde annotation $\tilde{f}$ is only used in the type-and-effect system; it does not have runtime semantics. That is the only annotation that is required in the source code.

The semantics is the same as the language in section 2, we thus omit the details.

5.2 Effects and Potentials

As seen from Figure 5, the definition of potentials (π) and effects (ϕ) depends on *roots* (β). Roots are the shortest path that represents an alias of a value that may not be transitively initialized. There are three roots in the system:

- $C.this$ represents an alias of *this* inside class C.
- $warm[C]$ represents an alias of a value of class C, all fields of which are assigned, but it may not be transitively initialized.
- *cold* represents a value whose initialization status is unknown. It is used to represent the potentials of cold class parameters. Field access or method calls on such an object is forbidden.

Potentials (π) represent aliasing information. They extend roots with field aliasing $\beta.f$ and method aliasing $\beta.m$. Field aliasing $\beta.f$ represents aliasing of the field f of β, while method aliasing $\beta.m$ represents aliasing of the return value of the method m with the receiver β.

Effects (ϕ) include field accesses, method calls and promotions of possibly uninitialized values. A promotion effect is represented with $\pi\uparrow$, which enforces that the potential π is transitively initialized. The field access effect $\beta.f!$ means that the field f is accessed on β. The method call effect $\beta.m\Diamond$ means that the method m is called on β.

There are three helpers for the creation of potentials and effects:

- Field selection: $select(\Pi, f)$
- Method call: $call(\Pi, m)$
- Class instantiation: $init(C, \overline{\hat{f}_i = \Pi_i})$

They are used in expression typing to summarize the potentials and effects of expressions. A key to understand the definitions is that the promotion effect $\pi\uparrow$ is the same as saying that π should be hot; and the empty set of potentials means that the result is hot.

Bounded Length. To make sure that the domain of effects and potentials is finite, the current system restricts the maximum length of potentials and effects to be two. In the implementation (Section 6), the maximum length of effects is 3. The bound is chosen in order to support calling methods on inner class instances, which is relatively common in Scala.

If the length of potentials exceeds the limit, the system checks that the potential is *hot* by producing a promotion effect. This can be seen from the last line of the definitions of the helper methods *select* and *call*.

Limiting the length will lead to incompleteness relative to the type system presented in Section 3. It does not pose a problem in practice (Section 7), due to the fact that fields usually hold hot values and methods return hot values. On the other hand, if it becomes an issue, the user may write explicit type annotations and the inference system can be extended to take advantage of the explicit type annotations.

Potentials and Effects

$$
\begin{array}{rcll}
T & ::= & C \mid D \mid E \mid \cdots & \text{type} \\
\beta & ::= & C.this \mid warm[C] \mid cold & \text{root} \\
\pi & ::= & \beta \mid \beta.f \mid \beta.m & \text{potential} \\
\Pi & ::= & \{\, \pi_1, \pi_2, \cdots \,\} & \text{potentials} \\
\phi & ::= & \pi\uparrow \mid \beta.f! \mid \beta.m\Diamond & \text{effect} \\
\Phi & ::= & \{\, \phi_1, \phi_2, \cdots \,\} & \text{effects} \\
\Omega & ::= & \{\, f_1, f_2, \cdots \,\} & \text{fields} \\
\Delta & ::= & \overline{f_i \mapsto (\Phi_i, \Pi_i)} & \text{field summary} \\
\mathcal{S} & ::= & \overline{m_i \mapsto (\Phi_i, \Pi_i)} & \text{method summary} \\
\mathcal{E} & ::= & \overline{C \mapsto (\Delta, \mathcal{S})} & \text{effect table}
\end{array}
$$

Select

$$
\begin{array}{rcl}
select(\Pi, f) & = & \Pi.map(\pi \Rightarrow select(\pi, f)).reduce(\oplus) \\
select(\beta, \tilde{f}) & = & (\emptyset, \{cold\}) \\
select(\beta, \hat{f}) & = & (\emptyset, \emptyset) \\
select(\beta, f) & = & (\{\beta.f!\}, \{\beta.f\}) \text{ where } \beta \neq cold \\
select(cold, f) & = & (\{cold\uparrow\}, \emptyset) \\
select(\pi, f) & = & (\{\pi\uparrow\}, \emptyset) \text{ where } \pi = \beta.f \text{ or } \pi = \beta.m
\end{array}
$$

Call

$$
\begin{array}{rcl}
call(\Pi, m) & = & \Pi.map(\pi \Rightarrow call(m, \pi)).reduce(\oplus) \\
call(\beta, m) & = & (\{\beta.m\Diamond\}, \{\beta.m\}) \text{ where } \beta \neq cold \\
call(cold, m) & = & (\{cold\uparrow\}, \emptyset) \\
call(\pi, m) & = & (\{\pi\uparrow\}, \emptyset) \text{ where } \pi = \beta.f \text{ or } \pi = \beta.m
\end{array}
$$

Init

$$
\begin{array}{rcl}
init(C, \overline{\hat{f_i} = \Pi_i}) & = & (\cup\overline{\Pi_{k\neq j}}\uparrow, \{warm[C]\}) \text{ if } \exists \tilde{f_j}, \Pi_j \neq \emptyset \\
init(C, \overline{\hat{f_i} = \Pi_i}) & = & (\cup\overline{\Pi_i}\uparrow, \emptyset)
\end{array}
$$

Helpers

$$
\begin{array}{rcl}
\Pi\uparrow & = & \{\, \pi\uparrow \mid \pi \in \Pi \,\} \\
(A_1, A_2) \oplus (B_1, B_2) & = & (A_1 \cup B_1, A_2 \cup B_2)
\end{array}
$$

Fig. 5. Potentials and Effects

5.3 Expression Typing

Expression Typing $\boxed{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)}$

$$\frac{x : D \in \Gamma}{\Gamma; C \vdash x : D \,!\, (\emptyset, \emptyset)} \quad \text{(T-Var)}$$

$$\Gamma; C \vdash \mathit{this} : C \,!\, (\emptyset, \{C.\mathit{this}\}) \quad \text{(T-This)}$$

$$\frac{\Gamma; C \vdash e : D \,!\, (\Phi, \Pi) \qquad (\Phi', \Pi') = \mathit{select}(\Pi, f) \qquad E = \mathit{fieldType}(D, f)}{\Gamma; C \vdash e.f : E \,!\, (\Phi \cup \Phi', \Pi')} \quad \text{(T-Sel)}$$

$$\frac{\begin{array}{c}\Gamma; C \vdash e_0 : E_0 \,!\, (\Phi, \Pi) \qquad \overline{\Gamma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \\ (\overline{x_i{:}E_i}, D) = \mathit{methodType}(E_0, m) \qquad (\Phi', \Pi') = \mathit{call}(\Pi, m)\end{array}}{\Gamma; C \vdash e_0.m(\overline{e}) : D \,!\, (\Phi \cup \overline{\Phi_i} \cup \overline{\Pi_i}\!\uparrow \cup\, \Phi', \Pi')} \quad \text{(T-Call)}$$

$$\frac{\overline{\hat{f_i}{:}E_i} = \mathit{constrType}(C) \qquad \overline{\Gamma; C \vdash e_i : E_i \,!\, (\Phi_i, \Pi_i)} \qquad (\Phi', \Pi') = \mathit{init}(C, \overline{\hat{f_i} = \Pi_i})}{\Gamma; C \vdash \mathit{new}\ C(\overline{e}) : C \,!\, (\cup\overline{\Phi_i} \cup \Phi', \Pi')} \quad \text{(T-New)}$$

$$\frac{\begin{array}{c}\Gamma; C \vdash e_0 : E_0 \,!\, (\Phi_0, \Pi_0) \qquad E_1 = \mathit{fieldType}(E_0, f) \\ \Gamma; C \vdash e_1 : E_1 \,!\, (\Phi_1, \Pi_1) \qquad \Gamma; C \vdash e_2 : E_2 \,!\, (\Phi_2, \Pi_2)\end{array}}{\Gamma; C \vdash e_0.f = e_1; e_2 : E_2 \,!\, (\Phi_0 \cup \Phi_1 \cup \Pi_1\!\uparrow \cup \Phi_2, \Pi_2)} \quad \text{(T-Block)}$$

Fig. 6. Expression Typing

Expression typing (Figure 6) has the form $\Gamma; C \vdash e : D \,!\, (\Phi, \Pi)$, it means that the expression e in class C under the environment Γ, can be typed as D, and it produces effects Φ and has the potentials Π. Generally, when typing an expression, the effects of sub-expressions will **accumulate**, while potentials may be **refined** (via selection), **promoted** (used as arguments to methods).

The definitions assume several helper methods, such as $\mathit{fieldType}(C, f)$, $\mathit{methodType}(C, m)$ and $\mathit{constrType}(C)$, to look up in class table Ξ the type, respectively, of field $C.f$, of method $C.m$ and of the constructor of C.

5.4 Definition Typing

Definition typing (Figure 7) defines how programs, classes, fields and methods are checked. The checking happens in two phases:

(1) *first phase*: conventional type checking is performed and effect summaries are computed;
(2) *second phase*: effect checking is performed to ensure initialization safety.

The two-phase checking is reflected in the typing rule T-Prog. To type check a program $(\overline{C}, D)$, first each class is type checked separately for well-typing and the effect summary for fields Δ_c and methods $\mathcal{S}_c$ is computed using class typing $\Xi \vdash C \,!\, (\Delta, \mathcal{S})$. The result of class typing is stored in the effect table $\mathcal{E}$, which is then used for modular effect checking of each class. Effect checking is performed modularly on each class with the help of the effect table $\mathcal{E}$. The typing rule T-Prog also checks that the entry class D is well-typed.

Program Typing $\boxed{\vdash \mathcal{P}}$

$$\frac{\begin{array}{ccc}\Xi = \overline{C \mapsto C} & \Xi(D) = \mathit{class}\ D\ \{\ \mathit{def}\ \mathit{main} : T = e\ \} & \emptyset; D \vdash e : T\ !\ (\Phi, \Pi) \\ \overline{\Xi \vdash C\ !\ (\Delta_c, \mathcal{S}_c)} & \mathcal{E} = \overline{C \mapsto (\Delta_c, \mathcal{S}_c)} & \overline{\Xi; \mathcal{E} \vdash C}\end{array}}{\vdash (\overline{C}, D)} \quad \text{(T-Prog)}$$

Effect Checking $\boxed{\Xi; \mathcal{E} \vdash C}$

$$\frac{(\Delta, \_) = \mathcal{E}(C) \qquad \overline{(\Phi, \_) = \Delta(f_i) \qquad \mathcal{E}; C^{\{ f_1, \cdots, f_{i-1} \}} \vdash \Phi}}{\Xi; \mathcal{E} \vdash \mathit{class}\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}} \quad \text{(T-Check)}$$

Class Typing $\boxed{\Xi \vdash C\ !\ (\Delta, \mathcal{S})}$

$$\frac{\overline{\Xi; C \vdash \mathcal{F}_i\ !\ (\Phi_i, \Pi_i)} \qquad \Delta = \overline{f_i \mapsto (\Phi_i, \Pi_i)} \qquad \overline{\Xi; C \vdash \mathcal{M}_i\ !\ (\Phi_i, \Pi_i)} \qquad \mathcal{S} = \overline{m_i \mapsto (\Phi_i, \Pi_i)}}{\Xi \vdash \mathit{class}\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \}\ !\ (\Delta, \mathcal{S})} \quad \text{(T-Class)}$$

Field Typing $\boxed{\Xi; C \vdash \mathcal{F}\ !\ (\Phi, \Pi)}$

$$\frac{\emptyset; C \vdash e : D\ !\ (\Phi, \Pi)}{\Xi; C \vdash \mathit{var}\ f : D = e\ !\ (\Phi, \Pi)} \quad \text{(T-Field)}$$

Method Typing $\boxed{\Xi; C \vdash \mathcal{M}\ !\ (\Phi, \Pi)}$

$$\frac{\overline{x{:}D}; C \vdash e : E\ !\ (\Phi, \Pi)}{\Xi; C \vdash\ \mathit{def}\ m(\overline{x{:}D}) : E = e\ !\ (\Phi, \Pi)} \quad \text{(T-Method)}$$

Fig. 7. Definition Typing

When type checking a class, the rule T-Class checks that the body fields and methods are well-typed, and the associated effects and potentials are computed. The effects and potentials associated with a field are the effects and potentials of its initializer (the right-hand-side expression). The effects and potentials associated with a method are the effects and potentials of the body expression of the method. The effect summaries are used during the second phase in T-Check, where it checks that given the already initialized fields, the effects on the right-hand-side of each field are allowed.

The typing rule T-Field checks the right-hand-side expression e in an empty typing environment, as there are no variables in a class body (class parameters are fields of their defining class). In the typing rule T-Method, the method parameters $\overline{x : D}$ are used as the typing environment to check the method body.

Propagate Potentials $\boxed{\mathcal{E} \vdash \pi \rightsquigarrow \Pi}$

$$\mathcal{E} \vdash \beta \rightsquigarrow \emptyset$$

$$\frac{(\Delta, \_) = \mathcal{E}(C) \qquad (\_, \Pi) = \Delta(f)}{\mathcal{E} \vdash C.this.f \rightsquigarrow \Pi} \qquad \frac{\mathcal{E} \vdash C.this.f \rightsquigarrow \Pi \qquad \Pi' = [C.this \mapsto warm[C]]\Pi}{\mathcal{E} \vdash warm[C].f \rightsquigarrow \Pi'}$$

$$\frac{(\_, \mathcal{S}) = \mathcal{E}(C) \qquad (\_, \Pi) = \mathcal{S}(m)}{\mathcal{E} \vdash C.this.m \rightsquigarrow \Pi} \qquad \frac{\mathcal{E} \vdash C.this.m \rightsquigarrow \Pi \qquad \Pi' = [C.this \mapsto warm[C]]\Pi}{\mathcal{E} \vdash warm[C].m \rightsquigarrow \Pi'}$$

Propagate Effects $\boxed{\mathcal{E} \vdash \phi \rightsquigarrow \Phi}$

$$\mathcal{E} \vdash \beta.f! \rightsquigarrow \emptyset \qquad \frac{(\_, \mathcal{S}) = \mathcal{E}(C) \qquad (\Phi, \_) = \mathcal{S}(m)}{\mathcal{E} \vdash C.this.m\Diamond \rightsquigarrow \Phi}$$

$$\frac{\mathcal{E} \vdash \pi \rightsquigarrow \Pi}{\mathcal{E} \vdash \pi\!\uparrow \rightsquigarrow \Pi\!\uparrow} \qquad \frac{\mathcal{E} \vdash C.this.m\Diamond \rightsquigarrow \Phi \qquad \Phi' = [C.this \mapsto warm[C]]\Phi}{\mathcal{E} \vdash warm[C].m\Diamond \rightsquigarrow \Phi'}$$

Closure

$$\frac{\Phi \subseteq \Phi' \qquad \forall \phi \in \Phi'. \mathcal{E} \vdash \phi \rightsquigarrow \Phi'' \implies \Phi'' \subseteq \Phi'}{\Phi^c = \Phi'}$$

Check $\boxed{\mathcal{E}; \Omega; C \vdash \Phi}$

$$\frac{\beta\!\uparrow \notin \Phi^c \qquad \forall C.this.f! \in \Phi^c.\ f \in \Omega}{\mathcal{E}; C^{\Omega} \vdash \Phi}$$

Fig. 8. Effect Checking

5.5 Effect Checking

The effect checking judgment $\mathcal{E}; C^{\Omega} \vdash \Phi$ (Figure 8) means that the effects Φ are permitted inside class C when the fields in Ω are initialized. It first checks that there is no promotion of *this* in the closure of the effects, as the underlying object is not transitively initialized, the promotion thus is illegal. Then it checks that each accessed field is in the set Ω, i.e., only initialized fields are used.

The closure of effects is presented in a declarative style for clarity, but it has a straight-forward algorithmic interpretation: it just propagates the effects recursively until a fixed-point is reached. The fixed-point always exists as the domain of effects and potentials is finite for any given program.

The main step in fixed-point computation is the propagation of effects and potentials. In effect propagation $\mathcal{E} \vdash \phi \rightsquigarrow \Phi$, field access $\beta.f!$ is an atomic effect, thus it propagates to the empty set. For a promotion effect $\pi\!\uparrow$, we first propagate the potential π to a set of potentials Π, and then promote each potential in Π. For a method call effect $C.this.m\Diamond$, it looks up the effects associated with the method from the effect table.

In potential propagation $\mathcal{E} \vdash \pi \rightsquigarrow \Pi$, root potentials like $C.this$ propagate to the empty set, as they do not contain *proxy* aliasing information in the effect table. For a field potential like $C.this.f$, it just looks up the potentials associated with the field f from the effect table. For a method potential $C.this.m$, it looks up the potentials associated with the method m from the effect table.

The soundness theorem says that a well-typed program does not get stuck at runtime.

Theorem 5.1 (Soundness). *If* $\vdash \mathcal{P}$, *then* $\forall k.\ [\![\mathcal{P}]\!](k) \neq Error$

The meta-theory takes the approach of step-indexed definitional interpreters [Amin and Rompf 2017]. Initialization safety is implied by soundness, as initialization errors will cause the program to fail at runtime. We refer the reader to the technical report for more details about the meta-theory [Liu et al. 2020].

6 IMPLEMENTATION

Based on the type-and-effect inference system, we implement an initialization system for Scala. The implementation is already integrated in the Scala 3 compiler [Odersky et al. 2013] and available to Scala programmers via the compiler option `-Ycheck-init`.

The implementation supports inner classes, first-class functions, traits and properties. Instantiation of inner classes is supported without any annotations, as the following example shows:

```
    class Trees {
      private var counter = 0
      class ValDef { counter += 1 }     // ok, counter is initialized
      class EmptyValDef extends ValDef
      val theEmptyValDef = new EmptyValDef
    }
```

To make the example above work, a warm potential in the system takes the form $warm(C, \pi)$, where C is the concrete class of the object, π is the potential for the immediate outer of C. The current version of the system only allows creating cyclic data structures via inner classes, passing *this* as arguments to new-expressions is disallowed. Supporting the usage requires the introduction of an annotation to the language thus involves a language improvement process, which we want to avoid in the initial version. We plan to support this in the next version following the solution outlined in the theory (Section 5).

To support first-class functions, we introduce the potential $Fun(\Phi, \Pi)$, where Φ is the set of effects to be triggered when the function is called, while Π is the set of potentials for the result of the function call. For example, it enables the following code, which is rejected in Swift:

```
class Rec {
  val even = (n: Int) => n == 0 || odd(n - 1)
  val odd = (n: Int) => n == 1 || even(n - 1)
  val flag: Boolean  = odd(6)
}
```

In functional programming, the recursive binding construct *letrec* may introduce similar initialization patterns as the code above. With the latest checker [Reynaud et al. 2018], OCaml still does not support the code below in the construct *let rec*:

```
let rec even n = if n = 0 then true else odd (x - 1)
    and odd n = if n = 0 then false else even (x - 1)
    and flag = odd 3
```

Naive extension of the type-and-effect system can easily lead to non-termination of effect checking in practice. This can be demonstrated by the following example:

Project	KLOC	W/K	W	X1	X2	X3	X4	A	B	C	D	E	F	G	H
DOTTY	106.0	0.73	77	742	447	146	350	7	16	2	32	0	3	4	13
INTENT	1.8	39.53	71	10	290	0	1	0	0	0	71	0	0	0	0
ALGEBRA	1.3	4.70	6	1	6	0	0	0	0	0	0	0	0	6	0
STDLIB213	43.6	0.62	27	231	104	8	99	14	0	4	2	0	1	6	0
SCALACHECK	5.5	1.08	6	39	70	6	83	0	0	0	6	0	0	0	0
SCALATEST	378.9	0.39	149	1037	718	18	664	0	0	8	114	0	8	19	0
SCALAXML	6.8	0.15	1	36	13	0	0	0	0	0	0	0	0	1	0
SCOPT	0.3	0.00	0	6	4	0	0	0	0	0	0	0	0	0	0
SCALAP	2.2	5.43	12	62	57	2	108	0	0	0	7	5	0	0	0
SQUANTS	14.1	0.00	0	9	0	0	0	0	0	0	0	0	0	0	0
BETTERFILES	2.8	0.00	0	17	1	0	0	0	0	0	0	0	0	0	0
SCALAPB	16.2	0.31	5	28	10	0	6	4	0	0	1	0	0	0	0
SHAPELESS	2.5	0.79	2	5	0	0	0	0	0	0	0	2	0	0	0
EFFPI	5.7	0.53	3	15	5	0	12	0	0	0	3	0	0	0	0
SCONFIG	21.8	0.60	13	70	43	0	8	13	2	2	0	0	1	6	2
MUNIT	2.7	1.13	3	32	73	1	13	0	0	0	2	0	0	0	1
SUM	612.1	0.61	375	2340	1841	181	1344	38	18	16	238	7	13	42	16

Fig. 9. Experiment result. The column W/K is the number of warnings per KLOC, and the column W is the number of warnings issued for the corresponding project. Other columns are explained in the text.

```
class B {
  class C extends B
  val c: C = new C
}
```

The code above involves an infinite sequence of constructor call effects of the form $\pi_i.init(C)$, where $\pi_0 = warm(C, this)$ and $\pi_i = warm(C, \pi_{i-1})$. We have to resort to a standard technique in abstract interpretation, widening [Cousot and Cousot 1991]. In the example above, we can stop the infinite sequence by widening π_i to $warm(C, cold)$.

One advantage of the type-and-effect system is that it integrates well with the compiler without changing the core type system. In contrast, integrating a type-based system in the compiler poses an engineering challenge, as the following example demonstrates:

```
class Knot {
  val self: Knot @cold = this
}
```

In the code above, the type of the field `self` depends on *when* we ask for its type. If it is queried during the initialization of the object, then it has the type `Knot @cold`. Otherwise, it has the type `Knot`. We do not see a principled way to implement the type-based solution in the Scala 3 compiler.

7 EVALUATION

We evaluate the implementation on a significant number of real-world projects, with zero changes to the source code. The experiment results are shown in Figure 9. The first three columns show the size of the projects and warnings reported for each project:

- **KLOC** - the number of lines of code (KLOC) in the project checked by the system
- **W/K** - the number of warnings issued by the system per KLOC
- **W** - the number of warnings issued by the system

We can see that for over 0.6 million lines of code, the system reports 375 warnings in total, the average is 0.61 warnings per KLOC. We can better interpret the data in conjunction with the following columns:

- **X1** - the number of field accesses on `this` during initialization
- **X2** - the number of method calls on `this` during initialization
- **X3** - the number of field accesses on *warm* objects during initialization
- **X4** - the number of method calls on *warm* objects during initialization

The data for the columns above are censused by the initialization checker, one per source location. Without type-and-effect inference, the system would have to issue one warning for each method call on `this` and warm objects [2], i.e., the counts in columns X1-X4 would all become warnings. This contributes more than 5700 warnings, a 15-fold increase in the number of warnings.

We manually analyzed all the warnings, and classified them into 8 categories:

- **A** - Use `this` as constructor arguments, e.g. `new C(this)`
- **B** - Use `this` as method arguments, e.g. `call(this)`
- **C** - Use inner class instance as constructor arguments, e.g. `new C(innerObj)`
- **D** - Use inner class instance as method arguments, e.g. `call(innerObj)`
- **E** - Use uninitialized fields as by-name arguments
- **F** - Access non-initialized fields
- **G** - Call external Java or Scala 2 methods
- **H** - others

The warnings in category **A** and **C** are related to the creation of cyclic data structures. From Section 5, we know such code patterns can be supported by declaring a class parameter to be *cold*. The current implementation does not support any annotations yet, we plan to introduce explicit annotations in the next version of the system.

Most of the warnings lie in the category **D**, which refer to cases like the following:

```scala
object Foo {
  case class Student(name: String, age: Int)
  call(Student("Jack", 30)            // should be OK, currently a warning
}
```

For the code above, our system currently issues a warning, as it only knows that the object created by `Student("Jack", 20)` is warm, while method arguments are required to be hot. Checking whether an inner class instance may be safely promoted to hot or not can be expensive if the inner class contains many fields and methods. However, it suggests that the system could be improved for common use cases that only involve small classes, such as the example above.

The category **E** refers to cases like the following, which is not supported currently:

```scala
def foo(x: => Int) = new A(x)
class A(init: => Int)
class Foo {
  val a: A = foo(b) // category E
  val b: Int = 100
}
```

As an over-approximation, we expect the warnings in category **F** are all false positives. However, to our delight, the system actually finds 8 true positives in ScalaTest, and one true positive in the Scala standard library. It also discovers two bugs in the Scala 3 compiler. We reported the bugs and they are already fixed.

[2]If we forget that non-private field accesses are also method calls in Scala.

The category **G** involves method calls on `this` in the constructor, but the target method is compiled by Java or the Scala 2 compiler. The category **H** invovles code that performs pattern matching on `this`, or calling methods on `cold` values. We discuss more about the experiment results in the appendix.

8 RELATED WORK

Our work takes inspiration from several milestone papers on the problem of initialization.

Fähndrich and Leino [2003] introduce raw types like $T^{\text{raw}(S)}$ — a value of such a type is possibly under initialization, and all fields up to the superclass S are initialized. Class fields may not hold raw values, thus it does not support creating cyclic data structures. To overcome the limitation, they introduce *delayed types* [Fähndrich and Xia 2007]. The system ensures that the initialization of objects forms stacked time regions.

Qi and Myers [2009] introduce a flow-sensitive type-and-effect system for initialization based on masked types. The system is expressive, however, it leaves open the problem of typestate polymorphism and type-and-effect inference. Our work can be seen as an attempt to address the problems.

Summers and Müller [2011] show that initialization of cyclic data structures can be supported in a light-weight, flow-insensitive type system. The system cleverly uses subtyping to achieve typestate polymorphism. However, it leaves open the design of a dataflow analysis that enables the usage of already initialized fields. Our work effectively addresses the problem.

There is another main difference: our system favors *perfect monotonicity*, while the freedom model favors *strong monotonicity*. There are design trade-offs in both approaches. In our case, perfect monotonicity enables us to remove the abstraction *unclassified* and it is easy to safely use already initialized fields in the constructor. In contrast, the freedom model enables assigning a free object to the field of another free object anywhere, while in our system it is only possible in the constructor at initialization points. More concretely, the following example is supported by the freedom model, but not by our system:

```
class A {
  m(this)
  var b: b = new B(this)
  def m(a: A @free): Unit = a.b = new B(a) // !!
}
class B(a: A @free)
```

The assignment `a.b = new B(a)` in the method `m` will be rejected by our system, as `new B(a)` is a value under initialization (it holds a reference to a free value `a`). In our system, it is only possible to assign hot values to fields of cold objects, while in the freedom model it is possible to assign non-committed values to fields of non-committed values. Our design is based on our experience with Scala projects, where an object rarely escapes from its constructor and has its fields initialized elsewhere. Summers and Müller [2011] have similar observations (Section 8.1).

The Checker Framework enables many useful checkers for various properties of Java programs [Ernst and Ali 2010]. In particular, it implements and extends the freedom model. One major extension is the introduction of the annotation `UnknownInitialization`, which is in the same spirit as `warm`. A difference is that `warm` in our type-based model enjoys transitivity — a warm object may in turn contain warm fields. The initialization model in Checker Framework does not enjoy this kind of transitivity enabled by `warm`, despite the introduction of 4 annotations: `Initialized`, `UnderInitialization`, `UnknownInitialization` and `NotOnlyInitialized`.

The initialization in X10 [Zibin et al. 2012] employs an inter-procedural analysis to ensure safe initialization, which removes the annotation burden required when calling final or private methods on *this*. However, the analysis algorithm is not presented in the paper. To call virtual methods on *this*, annotations are required on method definitions.

The Billion-Dollar Fix [Servetto et al. 2013] introduces a new linguistic construct *placeholders* and *placeholder types* to support initialization of circular data structures. The work is orthogonal to the current work, in that we are constrained from introducing new language constructs and semantics.

REFERENCES

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 666–679. http://dl.acm.org/citation.cfm?id=3009866

Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). 2013. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Lecture Notes in Computer Science, Vol. 7850. Springer. https://doi.org/10.1007/978-3-642-36946-9

Patrick Cousot and Radhia Cousot. 1991. Comparison of the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Actes JTASPEFL'91 (Bordeaux, France), October 1991, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings (Series Bigre)*, Michel Billaud, Pierre Castéran, Marc-Michel Corsini, Kaninda Musumbu, and Antoine Rauzy (Eds.), Vol. 74. Atelier Irisa, IRISA, Campus de Beaulieu, 107–110.

Joe Duffy. 2010. On partially-constructed objects. http://joeduffyblog.com/2010/06/27/on-partiallyconstructed-objects/.

Michael D. Ernst and Mahmood Ali. 2010. Building and using pluggable type systems. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 375–376. https://doi.org/10.1145/1882291.1882356

Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, Ron Crocker and Guy L. Steele Jr. (Eds.). ACM, 302–312. https://doi.org/10.1145/949305.949332

Manuel Fähndrich and K Rustan M Leino. 2003. Heap monotonic typestates. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO).*

Manuel Fähndrich and Songtao Xia. 2007. Establishing object invariants with delayed types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 337–350. https://doi.org/10.1145/1297027.1297052

Joseph Gil and Tali Shragai. 2009. Are We Ready for a Safer Construction Environment?. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 5653. Springer, 495–519. https://doi.org/10.1007/978-3-642-03013-0_23

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification, Java SE 8 Edition.

John Hogg, Doug Lea, Alan Cameron Wills, Dennis de Champeaux, and Richard C. Holt. 1992. The Geneva convention on the treatment of object aliasing. *OOPS Messenger* 3, 2 (1992), 11–16. https://doi.org/10.1145/130943.130947

Fengyun Liu, Ondrej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. Safe Initialization of Objects. (2020), 141. http://infoscience.epfl.ch/record/279970

John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and P. Mager (Eds.). ACM Press, 47–57. https://doi.org/10.1145/73560.73564

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

Martin Odersky et al. 2013. Dotty Compiler: A Next Generation Compiler for Scala. https://dotty.epfl.ch/.

Benjamin C. Pierce. 2002. *Types and programming languages.* MIT Press.

Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 53–65. https://doi.org/10.1145/1480881.1480890

Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. 2018. A right-to-left type system for mutually-recursive value definitions. *CoRR* abs/1811.08134 (2018). arXiv:1811.08134 http://arxiv.org/abs/1811.08134

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix - Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 7920. Springer, 205–229. https://doi.org/10.1007/978-3-642-39038-8_9

Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

Alexander J. Summers and Peter Müller. 2011. Freedom before commitment: a lightweight type system for object initialisation. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 1013–1032. https://doi.org/10.1145/2048066.2048142

Yoav Zibin, David Cunningham, Igor Peshansky, and Vijay A. Saraswat. 2012. Object Initialization in X10. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer, 207–231. https://doi.org/10.1007/978-3-642-31057-7_10

A DISCOVERED BUGS

As an over-approximation, we expect the warnings are all false positives. However, to our delight, our initialization system finds real bugs in high-quality projects, such as the Scala 3 compiler, Scala standard library and ScalaTest.

In the ScalaTest project, the checker reports 8 true positives [3]. The errors have similar forms, the following code demonstrates two of them:

```scala
sealed abstract class Fact {
  val isVacuousYes: Boolean
  val isYes: Boolean

  final def stringPrefix: String =
    if (isYes) {
      if (isVacuousYes) "VacuousYes" else "Yes"
    }
    else "No"
}

class Binary_&(left: Fact, right: Fact) extends Fact {
    val rawFactMessage: String = {
        // ...
        factDiagram(0)
    }

    val isYes: Boolean = left.isYes && right.isYes
    val isVacuousYes: Boolean = isYes && (left.isVacuousYes || right.isVacuousYes)

    override def factDiagram(level: Int): String = {
        stringPrefix
    }
}
```

The problem with the code above is that when we create an instance of `Binary_&`, it will call `factDiagram`, which in turn calls `stringPrefix`, where the properties `isYes` and `isVacuousYes` are used before they are initialized in the class `Binary_&`. Such errors never cause null-pointer exceptions, when they slip into a large code base, it will take significant efforts to debug.

The following code demonstrates a bug in the Scala 3 compiler [4]:

```scala
class Scanner(...) {
  val indentSyntax = ...
  // ...
  nextToken()   // the call indirectly reach the property indentSyntax
}

class LookaheadScanner(indent: Boolean = false) extends Scanner(...) {
  override val indentSyntax = indent
  // ...
```

[3]https://github.com/scalatest/scalatest/issues/1481
[4]https://github.com/lampepfl/dotty/issues/7660

```
}
```

Our checker reports the following error:

```
[warn] -- Warning: dotty/compiler/src/dotty/tools/dotc/parsing/Scanners.scala:885:34
[warn] 885 |    override val indentSyntax = indent
[warn]     |                                  ^
[warn]     |Access non-initialized field indentSyntax. Calling trace:
[warn]     | -> class LookaheadScanner(...) { [Scanners.scala:884 ]
[warn]     |  -> nextToken() [Scanners.scala:1323 ]
[warn]     |   -> if (isAfterLineEnd) handleNewLine(lastToken) [Scanners.scala:311 ]
[warn]     |    -> indentIsSignificant = indentSyntax [ Scanners.scala:484]
```

The problem is that when we create an instance of `LookaheadScanner`, the call `nextToken()` in the super class `Scanner` will reach the overridden property `indentSyntax`, which is not yet initialized in the sub-class.

The other bug found in the Scala 3 compiler is related to a subtle optimization of lazy value definitions in traits [5], which is not in accord with the language specification. Without the initialization checker, the bug would be latent longer in the compiler.

The bug in the Scala standard library [6] can be illustrated with the code below:

```
object Promise {
  val Noop = new Transformation[Nothing, Nothing](...)

  class Transformation[-F, T] (...) extends DefaultPromise[T]() with ... {
    def this(...) = this(...)
  }

  class DefaultPromise[T](initial: AnyRef) extends ... {
    def this() = this(Noop: AnyRef)
  }
}
```

The problem is that when we initialize the field `Noop`, it creates an instance of `Transformation`, which calls the super constructor in `DefaultPromise`, where `Noop` is accessed before initialization.

B CHALLENGING EXAMPLES

One design goal of the Scala 3 initialization system is to keep the core type system of the compiler intact. Consequently, we require that all arguments to methods are fully initialized, which is in line with good initialization practices. Otherwise, new types such as *T@cold* must be introduced in the language to handle safe method overriding.

Even if we manage to change the type system, it does not automatically solve the problem. This is demonstrated by the following example:

```
class Knot {
  val self: Knot @cold = this
}
```

In the code above, the type of the field `self` depends on *when* we perform the check. If it is checked during the initialization of the object, then it has the type `Knot @cold`. Otherwise, it has the type `Knot`. How to integrate this kind of types in the compiler is an engineering challenge.

[5]https://github.com/lampepfl/dotty/issues/7434
[6]https://github.com/scala/bug/issues/11979

The current implementation is based on a type-and-effect system. It elegantly lays on top of the type system, thus avoids the problems that a type-based solution would cause.

However, during the experiment we do encounter some reasonable code patterns that current implementation does not support. The following code about `LazyList` construction is one such example:

```scala
trait LazyList[A] { ... }
implicit class Helper[A](l: => LazyList[A]) {
  def #:: [B >: A](elem: => B): LazyList[B] = ...
}
class Test {
  val a: LazyList[Int] = 5 #:: b
  val b: LazyList[Int] = 10 #:: a
}
```

In the code above, inside the class `Test`, we use `b` (before it is initialized) as a by-name argument to initialize the field `a`. Similar code patterns also appear in by-name implicits [7].

To support the example above, the system has to support passing objects under initialization as arguments to methods and constructors. There is a chance to support the usage above without complicating the type system if we *restrict that the methods are effectively final*. The restriction removes the burden of overriding checks. Class constructors are inherently final, thus is not a problem.

However, the restriction cannot handle some use cases. The following code is a common pattern in the Scala 3 compiler to create cyclic type structures:

```scala
class RecType(parentExp: RecType => Type) {
  val parent = parentExp(this)
}
```

A solution based on types would change the type of `parentExp` to something like `RecType @cold =>Type @cold`. The solution requires changes to the core type system, thus is not feasible as we discussed above.

We can make the field `parent` `lazy` to to silence the warning about the escape of `this`. However, as compilers are performance-sensitive, we cannot do that due to the potential performance penalty with lazy fields. Currently, we have to resort to `@unchecked` for such cases.

Making a field `lazy` and adding the annotation `@unchecked` are currently the two ways to suppress warnings for complex initialization code. The lazy trick is a panacea with the slight danger of turning actual initialization errors into non-termination. On the other hand, drawing the line of when `@unchecked` should be used is a difficult language design decision. We expect the insights developed in the meta-theory about local reasoning will contribute to the decision process.

[7] https://docs.scala-lang.org/sips/byname-implicits.html

C SEMANTICS OF THE EXPERIMENTAL LANGUAGE

Program evaluation $\boxed{[\![(\overline{C}, D)]\!] = (l, \sigma)}$

$$\begin{array}{lcl}
[\![(\overline{C}, D)]\!] & = & [\![e]\!](\{\ l \mapsto (D, \emptyset)\ \}, \emptyset, l) \\
 & & \text{where } \Xi = \overline{C \longrightarrow C} \text{ and } l \text{ is a fresh location} \\
 & & \text{and } \Xi(D) = class\ D\ \{\ def\ main : T = e\ \}
\end{array}$$

Expression evaluation $\boxed{[\![e]\!](\sigma, \rho, \psi) = (l, \sigma')}$

$$\begin{array}{lcl}
[\![x]\!](\sigma, \rho, \psi) & = & (\rho(x), \sigma) \\
[\![this]\!](\sigma, \rho, \psi) & = & (\psi, \sigma) \\
[\![e.f]\!](\sigma, \rho, \psi) & = & (\omega(f), \sigma_1) \text{ where } (l_0, \sigma_1) = [\![e]\!](\sigma, \rho, \psi) \\
 & & \text{and } (\_, \omega) = \sigma_1(l_0) \\
[\![e_0.m(\overline{e})]\!](\sigma, \rho, \psi) & = & [\![e_1]\!](\sigma_2, \rho_1, l_0) \\
 & & \text{where } (l_0, \sigma_1) = [\![e_0]\!](\sigma, \rho, \psi) \\
 & & \text{and } (C, \_) = \sigma_1(l_0) \\
 & & \text{and } lookup(C, m) = def\ m(\overline{x{:}T}) : T = e_1 \\
 & & \text{and } (\overline{l}, \sigma_2) = [\![\overline{e}]\!](\sigma_1, \rho, \psi) \\
 & & \text{and } \rho_1 = \overline{x \mapsto l} \\
[\![new\ C(\overline{e})]\!](\sigma, \rho, \psi) & = & (l, \sigma_3) \\
 & & \text{where } (\overline{l}, \sigma_1) = [\![\overline{e}]\!](\sigma, \rho, \psi) \\
 & & \text{and } \sigma_2 = [l \mapsto (C, \emptyset)]\sigma_1 \text{ where } l \text{ is fresh} \\
 & & \text{and } \sigma_3 = init(l, \overline{l}, C, \sigma_2) \\
[\![e_1.f = e_2; e]\!](\sigma, \rho, \psi) & = & [\![e]\!](\sigma_3, \rho, \psi) \\
 & & \text{where } (l_1, \sigma_1) = [\![e_1]\!](\sigma, \rho, \psi) \\
 & & \text{and } (l_2, \sigma_2) = [\![e_2]\!](\sigma_1, \rho, \psi) \\
 & & \text{and } \sigma_3 = assign(l_1, f, l_2, \sigma_2)
\end{array}$$

Initialization

$$\begin{array}{lcl}
init(\psi, \overline{l}, C, \sigma) & = & [\![\overline{\mathcal{F}}]\!](\sigma_1, \psi) \\
 & & \text{where } lookup(C) = class\ C(\overline{\hat{f}{:}T})\ \{\ \overline{\mathcal{F}}\ \overline{\mathcal{M}}\ \} \\
 & & \text{and } \sigma_1 = assign(\psi, \overline{\hat{f}}, \overline{l}, \sigma) \\
[\![var\ f : D = e]\!](\sigma, \psi) & = & assign(\psi, f, l_1, \sigma_1) \text{ where } (l_1, \sigma_1) = [\![e]\!](\sigma, \emptyset, \psi)
\end{array}$$

Helpers

$$\begin{array}{lcl}
[\![\overline{e}]\!](\sigma, \rho, \psi) & = & fold\ \overline{e}\ (Nil, \sigma)\ f \text{ where} \\
 & & f\ (ls, \sigma_1)\ e = let\ (l, \sigma_2) = [\![e]\!](\sigma_1, \rho, \psi)\ in\ (l :: ls, \sigma_2) \\
[\![\overline{\mathcal{F}}]\!](\sigma, \psi) & = & fold\ \overline{\mathcal{F}}\ \sigma\ f \text{ where } f\ \sigma_1\ \mathcal{F} = [\![\mathcal{F}]\!](\sigma_1, \psi) \\
assign(\psi, f, l, \sigma) & = & [\psi \mapsto (C, [f \mapsto l]\omega)]\sigma \text{ where } (C, \omega) = \sigma(\psi) \\
assign(\psi, \overline{f}, \overline{l}, \sigma) & = & [\psi \mapsto (C, [\overline{f \mapsto l}]\omega)]\sigma \text{ where } (C, \omega) = \sigma(\psi)
\end{array}$$

Fig. 10. Big-step semantics, defined as a definitional interpreter.